Lockimine andmebaasides

madala taseme lock – vahendid lockimise implementeerimiseks

1. SYSV semafor, POSIX semafor (Linux > 2.6)

```
sem_t blah; /* shared memory */
sem_init(&blah, 1, 1) ==> sem=1 ==> 1
sem_wait(&blah); ==> sem=0 ==> 0
sem_post(&blah); ==> ++sem ==> 1

sem_trywait(&blah);
sem_timedwait(&blah, (struct timespec *) timeout);

gcc -lrt
```

2. Pthread mutex (kergekaalulisem alternatiiv MT programmidele fork() / SYSV semaforide asemel)

```
gcc -lpthread
```

Pthread mutexid ja condition variabled on igale protsessile unikaalsed

Condition variable: vahe pthread_mutex_lock() -iga on see, et järgmist locki ei saada mitte siis, kui teine thread teeb pthread_mutex_unlock(), vaid siis kui teine thread teeb pthread_cond_signal() või pthread_cond_broadcast() - ehk siis vahepeal võib olla toimunud palju iteratsioone.

Win32 API analoogid kahele ülaltoodule:

```
CreateMutex();
WaitForSingleObject(); /* or WaitForMultipleObjects() */
ReleaseMutex();
```

3. Spinlock

Väldib süsteemse (või mingi library poolt implementeeritud) API overheadi. Toetub atomaarsetele operatsioonidele.

```
lock:                       # The lock variable. 1 = locked, 0 = unlocked.
    dd      0

spin_lock:
    mov     eax, 1          # Set the EAX register to 1.

loop:
    xchg    eax, [lock]     # Atomically swap the EAX register with
                            #  the lock variable.
                            # This will always store 1 to the lock, leaving
                            #  previous value in the EAX register.

    test    eax, eax        # Test EAX with itself. Among other things, this will
                            #  set the processor's Zero Flag if EAX is 0.
                            # If EAX is 0, then the lock was unlocked and
                            #  we just locked it.
                            # Otherwise, EAX is 1 and we didn't acquire the lock.

    jnz     loop            # Jump back to the XCHG instruction if the Zero Flag is
                            #  not set, the lock was locked, and we need to spin.

    ret                     # The lock has been acquired, return to the calling
                            #  function.
```

```
spin_unlock:
    mov     eax, 0          # Set the EAX register to 0.

    xchg    eax, [lock]     # Atomically swap the EAX register with
                            #  the lock variable.

    ret                     # The lock has been released.
```

Spinlockil on mõte ainult mitme protsessoriga masinas, vastasel korral ei toimu spinnimise ajal mitte midagi.

Laiendus: katkesta spinlock, et teistele protsessidele CPU aega anda: loop counter, sched_yield(). Toimib, juhul kui luku taga ootamised ei ole sagedased.

spinlock ja atomaarsed operatsioonid

Probleemid spinlocki koodiga moodsates süsteemides:

- out-of-order execution

```
wait_loop:     cmp eax, sync_var
               jne wait_loop
```

kuna instruktsioonide täitmise kiirus on suurem, kui mälusiini kiirus, tekib alati hunnik lugemisoperatsioone, mis täidetakse suvalises järjekorras. sync_var muutumisel peab protsessor need operatsioonid tagasi rollima. Lahendus: Pentium 4 pause instruktsioon ootab järgmise instruktsiooni täitmisega umbes nii kaua, et teisel protsessoril oleks võimalik mällu uus väärtus kirjutada.

- false sharing

Väikesed tükid mälu võivad sattuda sama cache line peale. Kui meil on sync_var ja mingi tükk mälu, mida protsess modifitseerib sync_var-iga kaitstud koodiosas (a.k.a critical section), siis see põhjustab vajaduse sama cache line pidevalt mälust uuesti lugeda (spinlockis olevad protsessid nõuavad sync_var-ile shared accessi, critical sectionis olev protsess aga exclusive accessi). Lahendus: iga sünkroniseerimismuutuja ja nendega kaitstud data paigutamine eraldi cache reale.

Koodinäited:

```
/* Cache alignment */
struct syn_str { int s_variable; };
void *p = malloc ( sizeof (struct syn_str) + 127 );
syn_str * align_p = (syn_str *)( (((int) p) + 127) & -128 );


; Pentium 4 optimized spinlock
get_lock:      mov eax, 1
               xchg eax, A    ; Try to get lock
               cmp eax, 0     ; Test if successful
               jne spin_loop

critical_section:
               <critical section code>
               mov A, 0       ; Release lock
               jmp continue

spin_loop:     pause          ; Short delay
               cmp 0, A       ; Check if lock is free
               jne spin_loop
               jmp get_lock
continue:
```

```
/* spin loop after first try; Intel C compiler / Win32 */
for (;;) {
	for (int i=0; i < SPIN_COUNT; i++) {
		if ( (i & SPIN_MASK) == 0
			&& m_dwLock == UNLOCKED
			&& InterlockedExchange( &m_dwLock, LOCKED )== UNLOCKED)
			return;
#ifdef _X86_
		_mm_pause();
#endif
	}
	SleepForSleepCount( cSleeps++ );
}
```

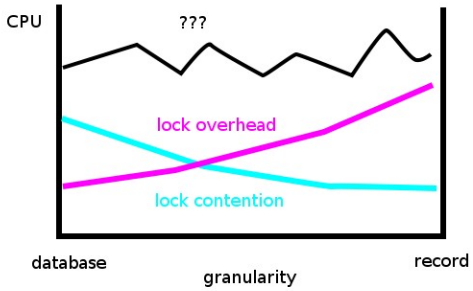
andmebaasi lock

Kahe threadi näide: mida peab andmebaasi progeja tegema?

	T1:	T2:
t0:	tmp1 := read(X)	
t1:		tmp2 := read(X)
t2:	tmp1 := tmp1 – 20	
t3:		tmp2 := tmp2 + 10
t4:	write tmp1 into X	
t5:		write tmp2 into X

Lukustamise juures on vaja teha erinevaid disainiotsuseid:

1. lock overhead / lock granularity tradeoff



2. andmete accessimise viisid – näide: kui iga read ja write sõltub ühe muutuja (rowcount) lockimisest, kujutab see endast loogiliselt terve tabeli locki

loogilise transaktsiooni tase

Anomaaliate liigid:

- dirty read
- phantom read
- non-repeatable read

ANSI/ISO SQL isolatsiooni kriteeriumid:

- SERIALIZABLE – transaktsioonide tulemus on ekvivalentne sellega, kui nad oleks järjestatud. Juhul kui querys on WHERE clause, tuleb kasutada range locki
- REPEATABLE READ – SELECT poolt tagastatud kirjed ei tohi muutuda. Uusi kirjeid võib ilmuda (phantom)
- READ COMMITTED – lubab non-repeatable readi, ehk siis andmed võivad ühe transaktsiooni sees kahe SELECT-i vahepeal muutuda
- READ UNCOMMITTED – lubab dirty readi

Need kriteeriumid ei ole eriti seotud populaarsete andmebaaside, nagu Oracle ja PostgreSQL isolatsioonimudelitega.

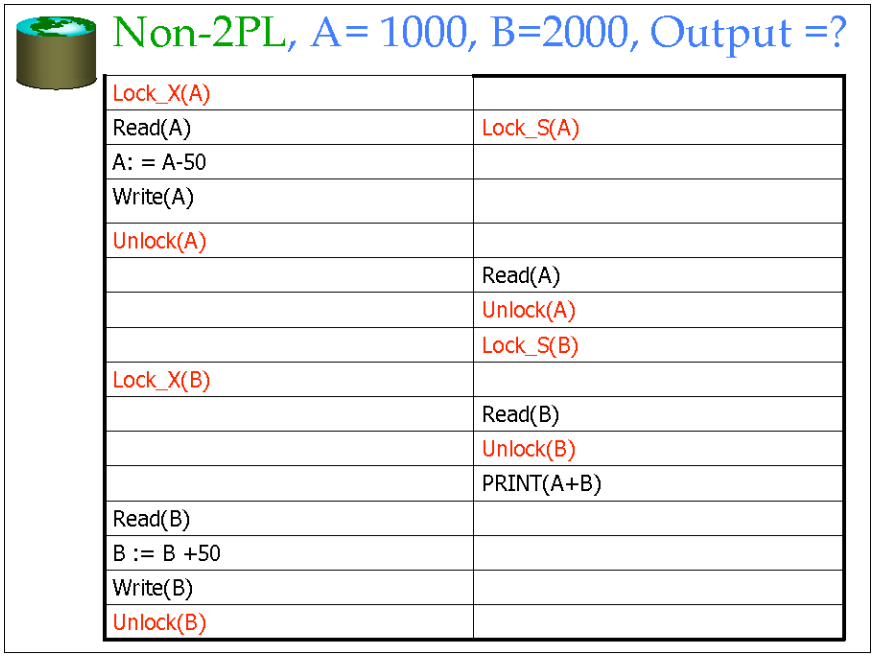
2-phase locking

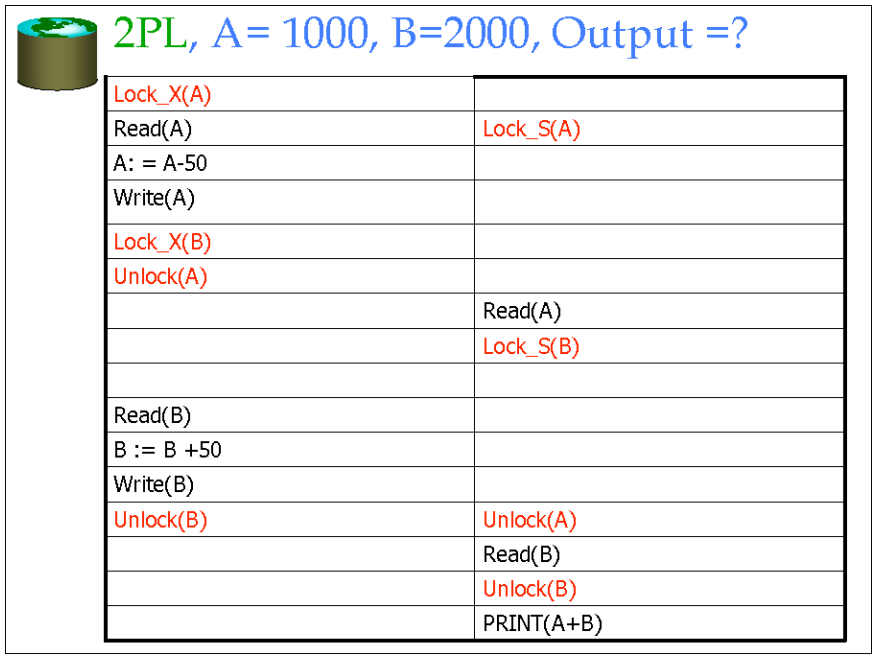
Põhineb lihtsatel printsiipidel:

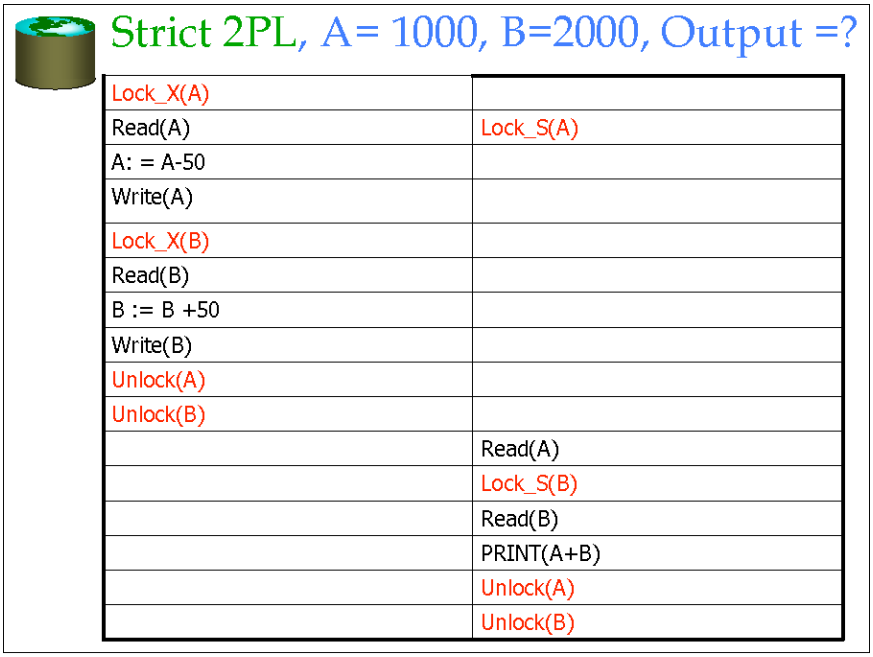
- iga transaktsioon peab lugemiseks omandama S (shared) locki ja kirjutamiseks X (exclusive) locki
- ükski transaktsioon ei tohi saada locki objektile, millele on võetud X lock
- ükski transaktsioon ei tohi saada X locki objektile, millele on võetud S lock
- 2PL: transaktsioon ei saa uusi locke, kui ta on vabastanud mõne locki
- Strict 2PL: kõik lockid vabastatakse üheaegselt peale transaktsiooni lõppemist

Kaks faasi on acquire phase ja release phase. Strict 2PL ei välista algoritmiliselt deadlockide tekkimist ja nõuab deadlock detection mehhanismi olemasolu.

Näited:







Antud näites 2PL ja Strict 2PL vahe on selles, et kui 2PL juures kirjutav transaktsioon oleks B updatemisel pidanud tagasi rollima, siis tuleb tagasi rollida ka lugev transaktsioon (A väärtus ei ole enam korrektne).

granularity levels

Miks vajalik:

- näiteks agregaatoperatsioonid jäiga isolatsiooni juures (MAX(age), SUM(balance))
- muud juhud, kus üks transaktsioon soovib lukustada suurt arvu ridu. Ainult reapõhiste lockide olemasolul tekitab hulga lisatööd.

Klassikaline hierarhia: database → table → page → record

Tasemetega lukustamisprotokoll:

- lisanduvad “intent” lukud, mis käivad madalama taseme kohta - IS (intent to get shared), IX (intent to get exclusive), SIX (IX + S on current level)
- et saada mingil tasemel S või IS lukku, peab kõrgemal tasemel omama IS või IX lukku
- et saada mingil tasemel X, IX või SIX lukku, peab kõrgemal tasemel omama IX või SIX lukku

Luku saamine samal tasemel (soovitud lukk / teise transaktsiooni poolt hoitav lukk):

	IS	IX	SIX	S	X
IS	√	√	√	√	-
IX	√	√	-	-	-
SIX	√	-	-	-	-
S	√	-	-	√	-
X	-	-	-	-	-

ilma lockimata lähenemine (optimistic locking)

- optimistic concurrency control (Kung-Robinson)

Põhineb transaktsioonide jaotamisel Read / Validate / Write faasidesse. Ainult lugemisoperatsioone tegevad transaktsioonid toimuvad takistamatult. Iga transaktsiooni juures luuakse ReadSet ja WriteSet. Validate faasi algoritm minimaalsel kujul: juhul kui ReadSet(Tk) ∩ WriteSet(Tj) ≠ ∅ siis Tk tuleb uuesti käivitada, sealjuures Tj on kõik transaktsioonid, mis lõpetasid oma Write faasi peale Tk käivitamist.

(vt. ka Agrawal, Carey, Livny 1987)

- timestamp CC

kolm timestampi – transaktsiooni timestamp TS(T), objekti lugemise timestamp RTS(O), objekti kirjutamise timestamp WTS(O)

lugemine:

```
if TS(T) < WTS(O)
        restart T
else
        read
        RTS(O):=max(RTS(O), TS(T))
```

kirjutamine:

```
if TS(T) < RTS(O)
        restart T
else if TS(T) < WTS(O)
        do nothing
else
        write
        WTS(O) := TS(T)
```

- multiversion CC (Oracle)

Kasutatakse jällegi timestampe TS(T), WTS(O_i) ja RTS(O_i). Objekti versioonid O_i on omavahel chainitud.

Lugemine:

```
find newest Oi where WTS(Oi) < TS(T)
```

kirjutamine:

```
find newest Oi where WTS(Oi) < TS(T)
if RTS(Oi)< TS(T)
        block all reads
        create Oj
        RTS(Oj) := WTS(Oj):= TS(T)
        unblock reads
else
        restart T
```

Eeldab, et vanu mittevajalikke versioone kustutatakse. Ei ole eriti ühilduv ANSI/ISO isolatsiooni standardiga, vt järgmist Oracle 9i näidet:

Time	Transaction 1	Transaction 2
1.	Begin transaction	
2.		Begin transaction
3.	Select available seats on flight ABC111. See seat 23F is the last seat available Reserve this seat.	
4.		Select available seats on flight ABC111. Also sees 23F as Oracle will go to the rollback segment to get the old version of that block.
5.	Commit Transaction.	
6.		Reserve this seat.
7.		Commit Transaction. Successful but now the flight is oversold.

Example IBM used to show incorrect logic in Oracle 9i version control.

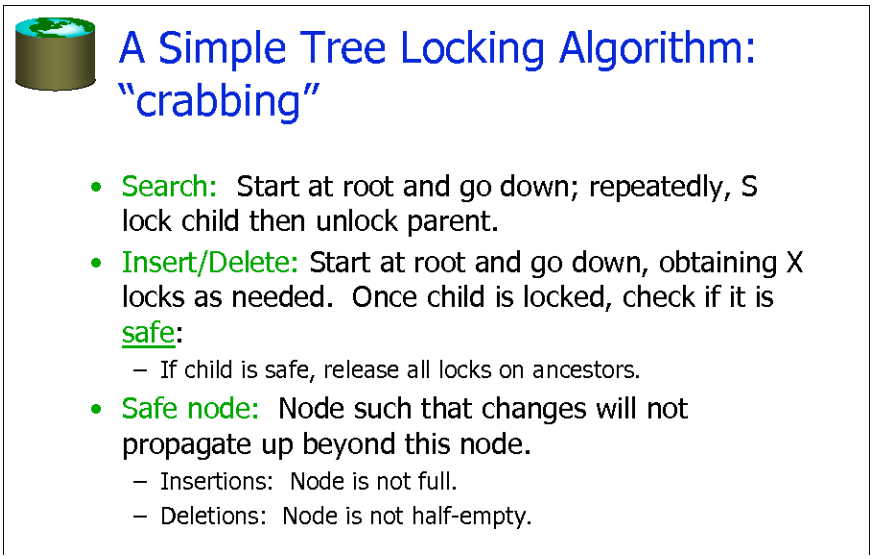
(Vt ka: http://www.ibphoenix.com/main.nfs?a=ibphoenix&page=ibp_expert4)

locking the index

1. Hash tabel – loogilisel tasemel triviaalne
2. B-tree.

Kõige olulisem eeldus, mis aitab paralleelsust tõsta: X locke tuleb võtta ainult nende node-de peale, mis insert/delete operatsiooni tagajärjel muutuvad.

Esmane lähenemine selle eelduse alusel:



Meetodid:

- Bayer-Schkolnick 1977, algoritm III

  Search: toimib nagu ülalpool
  Insert/Delete:
  - alusta root nodest
  - võta I (intent lock. Sama mis S, aga võib konverteerida X-iks).
  - Liigu puud mööda lehe poole, võttes child node jaoks I lukke. Juhul kui child node on "safe", vabasta parenti lock.
  - Kui leaf node on "safe", võta sellele X lukk ja update
  - Kui leaf node ei ole "safe", konverteeri kõik sel hetkel hoitavad I lukud X-ideks ning tee update.

- ARIES/KVL (MS SQL, paljud IBM tooted - DB2, MQ, Lotus Domino)
  võiks võtta edaspidiseks uurimiseks

Alternatiivsed lähenemised:

- ka btree puhul võib kasutada optimistic CC lähenemist (vt. Kung-Robinson 1981, peatükk 6)
- Kasuta tree indexit, kus insert/delete m6jutab ainult lehti - ISAM

index locking

Lähenemine ridade lockimisele tree indexi lockimise kaudu. Kui me võtame eelduseks, et pöördumine andmete poole käib alati indexi kaudu, siis indexi lockimine võimaldab järgnevat:

- index ise sassimineku eest kaitstud
- kuna indexit on niikuinii vaja lockida S ja X modes vastavalt andmetabelis toimuvale

operatsioonile, siis andmestruktuuride juures pole enam locki vaja.

- Kui agregaat- ja predikaatoperatsioonide juures õiged indexi node-d lockida, siis on vajalik ridade komplekt alati kaitstud (kaasa arvatud phantom read-ide vastu).

Praktiline algoritm: ARIES/KVL

Kui aega on: uuesti praktilise implementatsiooni juurde

Peene granulaarsusega lockimine shared mälus nõuab lockide (hash) tabeli tegemist (XXX: checki allikaid)?

Kui locki hoida andmerea juures, tuleb seda pad-ida, mis paisutab datat

```
Gavin Sherry <[EMAIL PROTECTED]> writes:
> What about padding the LWLock to 64 bytes on these architectures. Both P4
> and Opteron have 64 byte cache lines, IIRC. This would ensure that a
> cacheline doesn't hold two LWLocks.

I tried that first, actually, but it was a net loss.  I guess enlarging
the array that much wastes too much cache space.

                        regards, tom lane

---------------------------(end of broadcast)---------------------------
TIP 2: Don't 'kill -9' the postmaster
```

Shared locki implementeerimine spinlockina

Transaktsioonide implementimise juures (2PL) läheb critical section väga suureks

Kontrolltööks vajalik

- traditsioonilised lock semaforid jne
- spinlockid:
  - atomaarsus
  - ootamise peale prose mitteraiskamine
  - cache (eraldi 128 baidises ruumis)
- lockimise valtimine voi minimeerimine (optimistic locking, multiversion concurrency)
- 2phase locking
- btree locking

Viited

1. Using Spin-Loops on Intel Pentium 4 Processor and Intel Xeon Processor. Intel Corp. 2001
2. Berkeley CS186 kursus, http://inst.eecs.berkeley.edu/~cs186/archives.html
3. On Optimistic Methods for Concurrency Control. Kung, Robinson 1981
4. Concurrency Control Performance Modeling: Alternatives and Implications. Agrawal, Carey ja Livny 1987
5. http://www.firebirdsql.org/doc/whitepapers/fb_vs_ibm_vs_oracle.htm (Interbase/Firebird)
6. Concurrency of Operations on B-Trees. Bayer, Schkolnick 1977
7. ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes. C.Mohan 1990